Including Group-By in Query Optimization

Surajit Chaudhuri Kyuseok Shim*
Hewlett-Packard Laboratories
Palo Alto, CA 94304
chaudhuri@hpl.hp.com, shim@cs.umd.edu

Abstract

In existing relational database systems, processing of group-by and computation of aggregate functions are always postponed until all joins are performed. In this paper, we present transformations that make it possible to push group-by operation past one or more joins and can potentially reduce the cost of processing a query significantly. Therefore, the placement of group-by should be decided based on cost estimation. We explain how the traditional System-R style optimizers can be modified by incorporating the *greedy conservative* heuristic that we developed. We prove that applications of greedy conservative heuristic produce plans that are better (or no worse) than the plans generated by a traditional optimizer. Our experimental study shows that the extent of improvement in the quality of plans is significant with only a modest increase in optimization cost. Our technique also applies to optimization of `Select Distinct` queries by pushing down duplicate elimination in a *cost-based* fashion.

1 Introduction

Decision-support systems use the SQL operation of group-by and aggregate functions extensively in formulating queries. For example, queries that create *summary data* are of great importance in data warehouse applications. These queries partition data in several groups (e.g., in business sectors) and aggregate on some attributes (e.g., sum of total sales). A recent study of customer queries in DB2 [TM91] and other surveys [LCW93] have found that the group-by construct occurs in a large fraction of SQL queries used in decision-support applications. Therefore, efficient processing and optimization of queries with group-by and aggregation are of significant importance. Unfortunately, this problem has so far received little attention. For a single-block SQL query, the group-by operator is traditionally executed after all the joins have been processed. Such "two-phase" execution of aggregate queries is a consequence of the fact that the optimizers concentrate on selection, projection and join operators only. Conventional relational optimizers do *not* exploit the knowledge about the group-by clause in a query, beyond including the grouping columns in the list of *interesting orders* during join enumeration (discussed in Section 4). In this paper, we present significant new techniques for processing and optimization of queries with group-by.

*Work done while the author was visiting Hewlett-Packard Laboratories. Author's current address: Division of Research and Statistics, Federal Reserve Board, Washington, D.C. 20551.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

1.1 Motivating Application

Our examples are taken from a data warehouse application that analyzes trends in order placement. We have simplified the presentation for ease of exposition. A company has a set of business divisions. Each division belongs to a sector. Each product belongs to a division. For a given product, there is a fixed processing overhead for every order. An order is placed by a dealer. For every order, the name of the product and the amount and the date of sale are registered. Finally, for every dealer, the state and the street address are recorded. Thus, the relations and the attributes in the schema are:

```
Division(divid, sectorid)
Product(prodid, overhead, divid)
Order(orderid, prodid, dealerid, amount, date)
Dealer(dealerid, state, address)
```

1.2 Transformations

We make the key observation that since a group-by reduces the cardinality of a relation, an early evaluation of group-by could result in potential saving in the costs of the subsequent joins. We present an example that illustrates a transformation based on the above observation. An appropriate application of such a transformation could result in plans that are superior to the plans produced by conventional optimizers by an order of magnitude or more.

Example 1.1: Let us consider the query that computes the total sales for each sector of the company. Traditionally, this query is computed by taking the join among `Division`, `Product` and `Order`, subsequently doing a group-by on `sectorid` and computing `Sum(amount)`. However, the following alternative plan is possible. First, we group-by the `Order` relation on the attribute `prodid` before joining the relation with `Product`. In other words, we first compute the total sales for each product before the join. If each product has a large number of orders, then the above step may lead to a significant reduction in the size of the relation and hence in the cost of the subsequent join with `Product`. Next, we can group-by the resulting relation on `divid`. Intuitively, this computes the total sales achieved by each division. Finally, we join the resulting relation with `Division` and do a (residual) group-by on `sectorid`. ▮

In the above example, a single group-by in the traditional execution tree is replaced by *multiple group-by* operators in an equivalent execution plan and the grouping is done in stages, interleaved with join. In this paper, we will present other transformations as well. A simpler case of transformation is where the group-by operator in the given query is moved past joins, but is not broken up in stages. In such cases, the execution plan contains only one group-by but there are *multiple alternatives* in the execution plan where it may be placed. Furthermore, by maintaining the *count* of the tuples that were in the coalesced group, we can push down the group-by when neither of the above transformations apply. We will discuss such generalized transformations as well.

1.3 Optimization

The transformations that push down the group-by operator past joins must be judiciously applied depending on the query and the database. The following example shows that such transformations cannot be used blindly.

Example 1.2 : Consider Example 1.1 again. Assume that every product has a large number of orders and each division markets many products. For such statistics, the ordering of the joins in the traditional plan could be (from left to right) (`Division` $\bowtie$ `Product`) $\bowtie$ `Order`. The alternative execution plan suggested in Example 1.2 is attractive for such a database since the early group-by operations could result in a significant reduction in cardinality of the relations. However, the join ordering in the alternative plan is different from that in the traditional plan. In the former, the `Order` relation, after group-by, joins with `Product`. Subsequently, the resulting relation (possibly after another group-by) is joined with `Division`. This shows that the decision to push down group-by operations influences the join order. Also, an early application of a group-by operator is not always optimal. Thus, if there are only a few orders for every product and each division markets a few products only, then the alternative plan could perform worse than the traditional plan. ▮

The optimizer needs to identify where it can *correctly* place a group-by operator that can be evaluated early. Moreover, as the above example shows, its decision whether or not to push the group-by operator past the joins must be *cost-based*. Since the transformations affect the ordering of joins, their applications need to be considered in conjunction with the task of choosing a join order.

Since the optimization of queries with group-by cannot be treated in isolation, it is imperative that we incorporate our techniques in the framework of conventional optimizers. The System R style optimization algorithm [S*79] is used in commercial systems and so we will use that as the prototypical conventional optimizer. Integration of our optimization ideas in the conventional optimizer raises the following two issues. First, what effect does it have on the size of the *execution space*? Next, how do we extend the *search algorithm*, taking into account the trade-off between *cost of optimization* and the improvement in the *quality of plans*?

We note that *duplicate elimination* can be viewed as a special case of group-by where no aggregates are computed and the group-by is on all columns of the projection list. It has been recognized [DGK82, PHH92] that pushing down duplicate elimination past join could result in saving the cost of processing `Select Distinct` queries for Select-Project-Join expressions. However, as in pushing down group-by, the decision to push down duplicate elimination interacts with the ordering of joins. Therefore, our optimization algorithm for group-by applies to the problem of placing duplicate elimination operators in the execution plans for `Select Distinct` queries that may not even have explicit group-by clauses.

1.4 Related Work

In a recent paper [YL93], Yan and Larson identified a transformation that enables pushing the group-by past joins. Their approach is based on deriving two queries, one with and the other without a group-by clause, from the given SQL query. The result of the given query is obtained by joining the two queries so formed. Thus, in their approach, given a query, there is a *unique* alternate placement for the group-by operator. Observe that the transformation reduces the space of choices for join ordering since the ordering is considered only *within* each query. Our transformations vastly generalize their proposal and also avoids the problem of the reduced search space for join ordering. For example, the alternative execution suggested in Example 1.1 cannot be obtained by transformations in [YL93].

Prior work on group-by has addressed the problem of pipelining group-by and aggregation with join [D87, Kl82b] as well use of group-by to flatten nested SQL queries [K82, D87, G87, M92]. But, these problems are orthogonal to the problem of optimizing queries containing group-by that we are addressing in this paper.

1.5 Outline

Section 2 discusses the preliminary concepts and assumptions. In Section 3, we define the proposed transformations. Section 4 is devoted to the optimization algorithm. In Section 5, we discuss the experimental results using our implementation of the optimizer. The results in this section demonstrate that incorporating the transformations in the traditional cost-based optimizer is practical and results in significant improvement in the quality of the plan produced.

2 Preliminaries and Notation

2.1 Query

We will follow the operational semantics associated with SQL queries [DD93, ISO92]. We assume that the query is a single block SQL query, as below.

```
Select All <columnlist> AGG1(b1)..AGGn(bn)
From   <tablelist>
Where cond1 And cond2 ... And condn
Group By  col1,..colj
```

The `WHERE` clause of the query is a conjunction of simple predicates. SQL semantics require that `<columnlist>` must be among `col1,..colj`. In the above notation, `AGG1`..`AGGn` represent built-in SQL aggregate functions. In this paper, we will not be discussing the cases where there is a `HAVING` and/or an `ORDER BY` clause in the query. We will also assume that there are no nulls in the database. These extensions are addressed in [CS94].

We refer to columns in {`b1`, ..`bn`} as the *aggregating columns* of the query. The columns in {`col1`, ..`colj`} are called *grouping columns* of the query. The functions {`AGG1`, ..`AGGn`} are called the *aggregating functions* of the query. For the purposes of this paper, we will assume that every aggregate function has one of the following forms: `Sum(colname)`, `Max(colname)` or `Min(colname)`. Thus, we have excluded `Avg` and `Count` as well as cases where the aggregate functions apply on columns with the qualifier `Distinct`. In Section 3.4, we will discuss extensions of our techniques.

2.2 Extended Annotated Join Trees

An execution plan for a query specifies choice of access methods for each relation and an ordering of joins in the query. Traditionally, such an execution plan is represented syntactically as an *annotated join tree* [S*79] where the *root* is a group-by operation and each *leaf* node is a *scan* operation. An *internal* node represents a *join* operation. The annotations of a join node include the choice of the join method, as well as the selection conditions and the list of projection attributes. We assume that the selection conditions are evaluated and projections are applied as early as possible. The *optimization problem* is to choose a plan of least cost from its *execution space*. For optimization efficiency, the execution space is often restricted to be the class of *left-deep* join trees. These are annotated join trees where the right child of every internal node is a leaf.

The transformations that we propose introduce group-by operators as internal nodes. Therefore, we define *extended annotated join trees* which are annotated join trees except that a group-by may also occur as an internal node. Likewise, we can define *extended left-deep join trees*. These are trees subject to the same restrictions as the traditional left-deep join trees. For example, in Figure 1, tree (a) denotes a left-deep join tree, whereas trees (b) and (c) are extended left-deep join trees since the group-by occurs as an internal node in these trees. Finally, note that we mark the *scan* nodes by the name of the relation.

2.3 Group-By as an Operator

We assume that a group-by operator in an extended join tree is specified using the following annotations: (a) *Grouping Columns* (b) *Aggregating Columns*. The meaning of these annotations are analogous to the corresponding properties for the query (See Section 2.1). In reality, we need somewhat more elaborate annotations, including aggregating functions, but such details are not germane to our discussion here. We consider the question of determining the annotations of a group-by node if we place it immediately above a join or a scan node n.

Definition 2.1: *Join columns* of a node n are columns of n that participate in join predicates that are evaluated at ancestor nodes of n. *Required columns* of a node n are its join columns and the grouping columns of the query. ∎

Definition 2.2: *Candidate aggregating columns* of a node n are columns of n which are aggregating columns of the query but are *not* among required columns of the node. ∎

Observe that the columns of node n that are needed for subsequent processing are among required columns and candidate aggregating columns. In other words, these are the only columns that will be retained after projection is applied at node n. Definition 2.2 has an important subtlety. It excludes aggregating columns that will participate in future join predicates (and therefore, occur among required columns). Such columns can *not* be aggregated at node n, but must be retained until the join predicates are evaluated.

The annotations of a group-by node that is placed above a join or a scan node n will be defined as (1) The set of grouping columns is the set of required columns of n. (2) The set of aggregating columns is the set of candidate aggregating columns of n. Thus, given a node n, the specification of the group-by node that may be placed immediately above it is *unique*. Note that the annotations are such that all columns that can be *correctly* aggregated immediately after the node n are aggregated by the group-by following n.

Example 2.3: Let us consider tree (a) in Figure 1. The required columns of the scan node for `Order` are {`dealerid, prodid`}. The candidate aggregating column of the scan node for `Order` is `amount`. Indeed, tree (b) shows an extended execution tree where a group-by node with the above specifications has been placed above `Order`. Consider a variant of the query in tree (a) with the added selection condition `Order.amount` > `Product.overhead`. In such a case, the column `amount` of `Order` is among the required columns and cannot be aggregated (unlike in tree (b)) until the join with `Product` is completed. ∎

Several implementations of group-by are possible. For example, a group-by may be implemented by sorting the data on the grouping columns of the group-by node. Such an implementation is particularly useful if order-by is used in the SQL query in conjunction with group-by. Another popular alternative is based on hashing where the data stream is hashed on the grouping columns of the operator. Subsequently, the data is sorted within each bucket. Such an implementation ensures grouping but no ordering on the data. Also, scan and group-by operations can be combined by using index structures. A detailed discussion appears in [CS94].

2.4 Duplicate Elimination as an Operator

Duplicate elimination operation may be pushed down to *any* node of the annotated join trees. However, the challenge is to decide on applications of duplicate elimination in a cost-based way. Consider Select-Project-Join expressions that have the following form (note that we do not require such queries to have a group-by clause).

```
Select Distinct <columnlist>
From <tablelist>
Where cond1..condn
```

We observe that duplicate elimination is a special case of a group-by operator with no aggregating columns and columns in `<columnlist>` as its grouping columns. Therefore, optimization techniques developed in this paper apply directly for cost-based application of duplicate elimination. We refer the reader to [CS94] for further details.

3 Transformation of Annotated Trees

We say that two annotated join trees are *equivalent* for a given schema if they result in the same sets of answers over every database. In this section, we present transformations that generate an equivalent extended left-deep tree from a given left-deep tree. We show three transformations of increasing generality. In the first transformation, the extended left-deep tree is obtained by simply moving the group-by operator from the root to an internal node. In the second transformation, the single group-by in the left-deep tree is replaced by multiple group-by operators in the extended left-deep tree. The final transformation generalizes the previous transformations by maintaining the count of the tuples in the groups that are coalesced.

3.1 Invariant Grouping

The intuition behind the *invariant grouping* transformation is to identify one or more nodes n on the given left-deep tree such that we can obtain an equivalent extended left-deep tree from the given left-deep tree by moving the group-by operator to just above the node n. We call this transformation invariant since the annotations of the group-by operator in the given left-deep tree are not modified by the transformation.

We observe that in the traditional left-deep tree, the grouping columns and the aggregating columns of the group-by operator are the grouping columns and aggregating columns of the query respectively. Therefore, from Section 2.3, it follows that for this group-by operator to be placed immediately above a scan or a join node n, the candidate aggregating columns of n and the required columns of n must be the aggregating columns

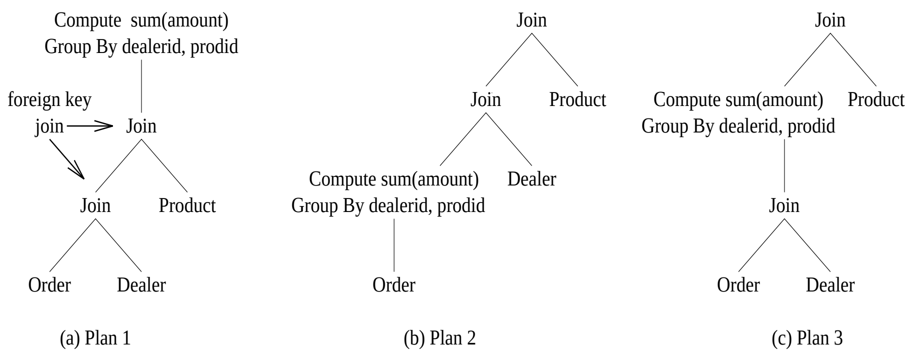

Figure 1: Invariant Grouping Transformation

of the query and the grouping columns of the query respectively. In that case, any two tuples that belong to different groups in an early evaluation of the group-by, must also belong to different groups in the answer as well. However, to ensure equivalence to the given left-deep tree, the subsequent joins should not result in more than one answer tuples that belong to the same group and need to be coalesced. The above condition is satisfied if the joins subsequent to the group-by are on columns that are *foreign keys*, as explained below. A foreign key can join with at most one tuple in the referenced relation and therefore contributes to at most one tuple in the answer. Therefore, if all joins subsequent to group-by are on foreign keys, then for every group formed by the early group-by, either every tuple that were in that group fails to produce any answer tuples, or every tuple produces exactly one answer tuple. Thus, the subsequent joins do not result in multiple tuples belonging to the same group in the answer relation and the extended annotated join tree is equivalent to the given left-deep tree. The following definition identifies such nodes n that have the invariant grouping property.

Definition 3.1: A node n of a given left-deep tree has the *invariant grouping* property if the following conditions are true:

1. Every aggregating column of the query is a candidate aggregating column of n.
2. Every join column of n is also a grouping column of the query.
3. For every join-node that is an ancestor of n, the join is an equijoin predicate on a foreign key column of n.

Theorem 3.2: *If a node n in a left-deep tree $\mathcal{T}$ has the invariant grouping property, then the extended left-deep tree $\mathcal{T}'$ obtained by moving the group-by node in $\mathcal{T}$ as the parent of n is equivalent to $\mathcal{T}$.*

From Definition 3.1, it follows that if a node n in a left-deep tree has the invariant grouping property, so do its ancestors. In other words, nodes that have the invariant grouping property form a chain and an equivalent extended annotated join tree may be obtained by moving the group-by operator to *any* one of the nodes in that chain. This observation is significant from the point of view of execution space of the optimizer. Finally, note that placing a group-by at more than one nodes with the invariant grouping property is computationally redundant.

Example 3.3: Consider the query that computes for each dealer in `California` and each product from `Telecom` division, the dollar value of the orders placed last month. Tree (a) in Figure 1 is a left-deep join tree for this query. We have not shown the predicates in the figure. Intuitively, the first join checks whether the dealer is in `California` and the second join checks whether the `divid` is `Telecom`. We observe that for the node `Order` in tree (a) the required columns ({`dealerid, prodid`}) are the grouping columns of the query, the candidate aggregate column ({`amount`}) is the aggregating column of the query and the future join predicates are equi-join on foreign keys (`dealerid`, `prodid` respectively). Therefore, `Order` has the invariant grouping property and so does its ancestors. From Theorem 3.2, it follows that tree (b) and tree (c) in Figure 1 are equivalent to tree (a). Intuitively, tree (b) corresponds to the alternative execution plan where for every product and every dealer, the sum of the orders is computed. Next, by joining the grouped relation with `Dealer` and `Product`, we retain only those groups that correspond to dealers in `California` and products that are marketed by `Telecom` division respectively. ▮

3.2 Simple Coalescing Grouping

The simplicity of invariant grouping transformation is remarkable since it enables us to move down the group-by operator without any modification to the annotations of the operator. However, applicability of invariant grouping requires satisfying conditions in Definition 3.1. *Simple coalescing* grouping generalizes invariant grouping by relaxing these conditions.

Like invariant grouping, simple coalescing grouping property is also useful for performing early group-by,

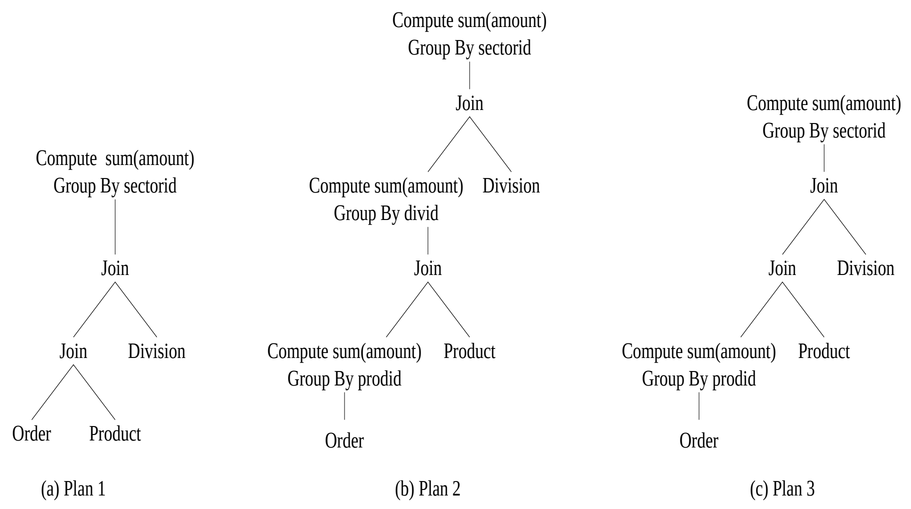

Figure 2: Simple Coalescing Grouping

but may require an additional group-by subsequently that coalesces multiple groups. Thus, as a consequence of application of simple coalescing, *multiple* group-by operators may replace the single group-by in a left-deep tree to yield an equivalent extended left-deep tree. As mentioned in the introduction, such a transformation could help obtain an execution plan that is superior to a plan produced by a traditional optimizer by an order of magnitude or more.

Let us begin by considering the possible effect of an early group-by on a node in an execution tree that does not satisfy condition (2) of Definition 3.1. In such a case, the tuple corresponding to the coalesced group may result in more than one tuples in the output relation that agree on the grouping columns of the query. For example, if g is the grouping column of the query but if (g, g') are the grouping columns of the node n, then, in the final result, all groups that agree on the same value of g must be coalesced. The effect of relaxing condition (3) of Definition 3.1 is similar. Therefore, to push down group-by when conditions (2) or (3) of Definition 3.1 is not true, we must be able to subsequently coalesce two groups that agree on the grouping columns. Fortunately, for the built-in SQL aggregate functions Agg (e.g., `Sum`) that we are considering (See Section 2.1), an aggregate over a bag of tuples may be computed from the aggregates computed from partitions of the bag:

$$Agg(S \cup S') = Agg(\{Agg(S), Agg(S')\}) \qquad (1)$$

where S and S' are arbitrary bags and $\cup$ denotes union of the bags. Thus, two groups can be coalesced by another application of group-by. We now formally define the property of simple coalescing grouping of a node in a left-deep tree.

Definition 3.4: Given a left-deep tree, a node n has the *simple coalescing grouping* property if all aggregating columns of the query are candidate aggregating columns of the node n. ∎

It follows that if a node n has the *simple coalescing grouping* property, then so does all its ancestors in the left-deep join tree. The following theorem says that placing group-by operators on one or more nodes along a chain of nodes with simple coalescing grouping property creates an extended left-deep tree which is equivalent to the given left-deep tree.

Theorem 3.5: *If a node n in a left-deep tree $\mathcal{T}$ has the simple coalescing grouping property, then the extended left-deep tree $\mathcal{T}'$ obtained from $\mathcal{T}$ by adding one or more group-by operator immediately above n or its ancestors is equivalent to $\mathcal{T}$.*

Thus, the theorem provides the optimizer with opportunities for alternate placements of the group-by operators. Notice that the extended left-deep tree contains the group-by node that was present in the given left-deep tree. Therefore, additional group-by nodes along the chain are *semantically* redundant. However, in contrast to invariant grouping, multiple applications of the group-by operator along a chain of nodes are not *computationally* redundant but results in "stagewise" grouping, as the following example illustrates.

Example 3.6: Let us reconsider Example 1.1 where the grouping and aggregation were done in a "stagewise" fashion. In Figure 2, the node `Order` does not satisfy the invariant grouping property since it does not have the column `sectorid`. . However, since the aggregating column of the query (`amount`) is also the candidate aggregating column of `Order`, it does satisfy the simple coalescing grouping property. From Theorem 3.5, it follows that tree (b) is equivalent to tree (a). Observe that the multiple applications of group-by are not computationally redundant. The successive group-by operators compute the sum of orders for each product, division and sector respectively. Notice that tree (c) provides a variant where no group-by node is placed following join with `Product`. However, tree (c) is equivalent to tree (a) as well. ∎

A node with the invariant grouping property is a special case of a node with the simple coalescing grouping property. Therefore, if we can recognize that a group-by operator introduced by a simple coalescing transformation is placed at a node with the invariant grouping property, we know that all subsequent group-by nodes, if any, are computationally redundant.

3.3 Generalized Coalescing Grouping

The scope of early grouping can be further extended if we do not require that all aggregating columns of the query are present in the node where an early group-by operator is placed.

Example 3.7: Let us consider the query that computes for every product, the overhead expenses incurred by that product due to orders in the last month. The traditional plan is obtained by taking the join between `Order` and `Product`. Subsequently, the result relation is grouped on `prodid` and the aggregate `Sum(overhead)` is computed. Intuitively, another alternative plan is to count the number of orders for each product and to multiply that number by `overhead` to obtain the total overhead for that product. Thus, we can do a group-by on the `Order` relation with `prodid` as the grouping column before taking the join with `Product`. The total overhead for sales of a `prodid` can be computed during the join by multiplying the the value of `overhead` with the number of `Order` tuples for that `prodid` that were coalesced into a single tuple by the application of group-by to `Order`. Note that `Order` does not satisfy the simple coalescing grouping property since the aggregating column of the query is `overhead`, which is not among the columns of `Order`. This example represents a significant family of queries where the referencing relation (`Order`) has a foreign key that connects it to the referenced relation and we aggregate on an attribute of the latter (`Product`). We refer to the such queries as *foreign relation aggregate* queries. ∎

To get an informal understanding of generalized coalescing, assume that the traditional plan is to apply an aggregate function Agg function on column $S.a$ after obtaining the result of $R \bowtie S$. However, we can group the relation R on its join columns to obtain the relation R' and then join it with S. Assume that a tuple s of S joins with a tuple r of R' that was obtained by coalescing N tuples of R. For simplicity, assume that r is the only tuple s joins with. In that case, the function Agg needs to be applied to the bag consisting of N copies of the tuple s (denoted Ns). Fortunately, for SQL built-in functions, the result of $Agg(Ns)$ can be derived from the result of $Agg(\{s\})$ as the following identities show:

```
Min(Ns) = s, Max(Ns) = s, Avg(Ns) = s
   Sum(Ns) = N*s, Count(*)(Ns) = N
```

Thus, by keeping the count N of the coalesced group, and using the identity, the result of aggregation over the coalesced group can be derived. We now present a formal account of generalized coalescing property. The transformation is applicable if every aggregate function Agg in the query has the property that there is a function f such that $Agg(Ns) = f(n, s)$.

Our proposal for supporting *generalized coalescing* requires extensions to the group-by and join operations, as discussed below.

- *General Group-By:* This is the augmented group-by operator, which in addition to grouping also ensures that the output stream has a special attribute `Group_Count` that carries the size of each coalesced group in the data stream. If the data stream already has a `Group_Count` attribute, then the values in that column are summed for each coalesced group.
- *Aggregate Join:* This is the same as the traditional join, except that it also handles a join with a stream that has the `Group_Count` attribute by using the identities on aggregates listed above.

Example 3.8: Example 3.7 illustrates a generalized coalescing transformation. In the second alternative plan, the group-by on `Order` that precedes the join with `Product` is a general group-by operator and introduces a `Group_Count` field for each coalesced group. Thus, if a certain product `P001` has 4 orders, then the `Group_Count` of the coalesced group of tuples in `Order` for `P001` is 4. The join between `Order` and `Product` is an aggregate join since it applies the identity `Sum(No) = N * o` where `N` is the value of the `Group_Count` column of the relation after performing a general group-by on `Order` and `o` is the value of `overhead` attribute. ∎

Theorem 3.9: *Given a left-deep tree $\mathcal{T}$, for every node n, an extended left-deep tree obtained from $\mathcal{T}$ by the following modifications is equivalent to $\mathcal{T}$.*

- *Replace every join node that is an ancestor of n by an aggregate join.*
- *Place one or more general group-by nodes immediately above (as the parent of) n or any of its ancestor nodes.*

The restriction that general group-by operators be placed only along a chain is not central and the generalization of the above theorem is straight-forward [CS94].

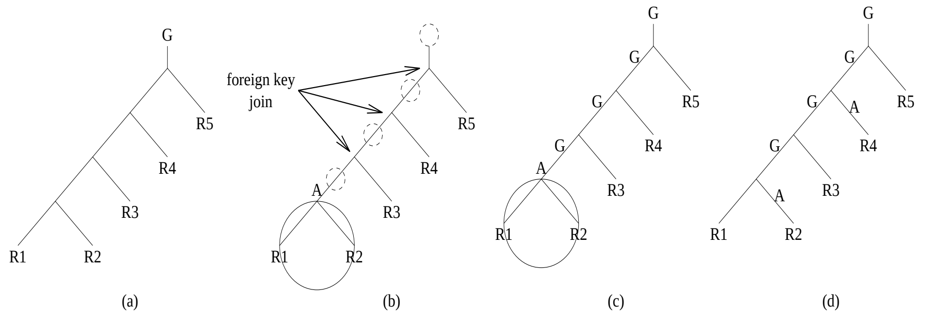

Figure 3: Comparison of Transformations

With respect to Theorem 3.9, it is worth noting that in case the functions on aggregating columns that appear in any ancestor of node n are among **Max**, **Min** and **Avg**, then the column **Group_Count** is not necessary. Also, aggregate joins on the chain may degenerate to traditional joins when certain conditions are true. Generalized coalescing is also useful in optimizing queries without group-by [CS94].

3.4 Discussion on Transformations

We have presented transformations that are of increasing generality and complexity. The applicability of the transformations are indicated in Figure 3. The subtree marked A denotes that all aggregating columns of the query occur in that subtree. A node marked G represents an application of group-by. Tree (a) represents a left-deep tree obtained by a traditional optimizer, tree (b) denotes a tree that has a chain of nodes (dotted circles) where invariant grouping property holds. The dotted circles indicate that the group-by operator may be placed in *only one* of the dotted nodes. Nodes in tree (c) have the simple coalescing property and thus multiple group-by operators may appear along a chain. Tree (d) shows that in a generalized coalescing transformation, a group-by operator may be placed even if not all aggregating columns of the query are candidate aggregating columns (e.g., in Figure 3).

Note that although we have presented the transformations in the context of left-deep trees, the transformations apply to bushy annotated join trees with obvious generalizations [CS94].

In Section 2.1, we said that we are considering only the SQL built-in functions **Max**, **Min** and **Sum**. While all three transformations that we proposed apply for these aggregate functions, the *generalized coalescing* transformation also handles **Avg** and **Count(*)** correctly by using **Group_Count**.

Let us also consider our earlier assumption that we require aggregates on columns to be qualified by **All**[1] (e.g., **Sum (All amount)**). This restriction is easily relaxed. First, note that the invariant grouping transformation is applicable on aggregates qualified by **Distinct** as well. Next, for simple and generalized coalescing, if there are aggregates in the query with **Distinct** qualifiers, we can consider the **Distinct** columns as part of the grouping columns of the query from the point of view of our transformations. Thus, while aggregates on these columns are not computed during an early grouping, we can still use our transformations for remaining aggregates that have the qualifier **All**.

We note that an invocation of a group-by operator is *redundant* if the set of required columns functionally determines the rest of the columns. In such a case, each group is a singleton. This is an important case that the optimizer must recognize in order to avoid generating redundant alternatives.

Finally, for a single block **Select Distinct** query without any group-by, the duplicate elimination operator may be pushed down to all join and scan nodes and their applications are not computationally redundant. Thus, we model duplicate elimination as a simple coalescing transformation. The condition for simple coalescing grouping property is trivially satisfied for such queries for all nodes in any left-deep tree since there are no aggregating columns.

4 Optimization Algorithm

The transformations of Section 3 translate an execution tree into an equivalent extended execution tree. As Example 1.2 shows, the decision whether to apply a transformation or not must be made by the optimizer in a *cost-based* fashion. However, such choice also *interacts with join ordering*. Therefore, we need to consider what modifications are needed to the algorithm [S*79] that chooses an optimal join order. The algorithm described in this section incorporates the invariant grouping and the simple coalescing transformations over extended left-deep join trees. Extensions to incorporate generalized coalescing are described in [CS94].

4.1 Traditional Approach

The aim of the optimizer is to produce an execution plan of least cost from a given execution space. Traditionally, the execution space has been limited to

[1] This is the default specification in SQL.

left-deep join trees (See Section 2.2) which correspond to linear orderings of joins. The well-known algorithm [S*79] to choose an optimal linear ordering of joins in the query uses dynamic programming. A query (say Q) is viewed as a *set* of relations (say, $\{R_1..R_n\}$) which are *sequenced* during optimization to yield a join order. The optimization algorithm proceeds stagewise, producing optimal plans for subqueries in each stage. Thus, for Q, at the ith stage ($2 \leq i \leq n$), the optimizer produces optimal plans for all subqueries of Q of *size* i (i.e., subqueries that consist of join of i relations). Consider a subquery Q' of size $i+1$. The steps in the function *Enumerate* are followed to find its optimal plan.

Function *Enumerate*

```
1. for all $R_j, S_j$ s.t $Q' = S_j \cup \{R_j\}$ do
2.     $p_j := joinPlan(optPlan(S_j),\ R_j)$
3. end for
4. $optPlan(Q') := MinCost_j(p_j)$
```

In steps 1 and 2, all possible ways in which a plan for Q' can be constructed by extending an optimal plan for a subquery of size i (i.e., S_j) are considered. Note that R_j is the remaining relation that occurs in Q'. The function *joinPlan* creates the plan for joining a relation with another intermediate (or a base) relation. Thus, access methods and choice of join algorithms are considered in *joinPlan*. In step 4, *MinCost* compares the plans constructed for Q' and picks a plan with the least cost. It can be shown that the above algorithm is optimal with respect to the execution space of left-deep trees. We refer the reader to [S*79, GHK92] for more details. The soundness of the above enumeration algorithm relies on the cost model satisfying *principle of optimality* [CLR90]. Thus, violation to the principle of optimality requires further extensions. Such violation can occur because presence of an appropriate order on relations can help reduce the cost of a subsequent sort-merge join since the sorting phase is not required. For example, consider the query $\{R_1, R_2, R_3\}$ with predicates $R_1.a = R_2.b$ and $R_2.b = R_3.c$. Let P be a suboptimal plan for $\{R_1, R_2\}$ that is sorted on the join column b (e.g., sort-merge was the join method in P). Since b is also the join column in the subsequent join with R_3, the ordering on P may lead to an optimal plan for $\{R_1, R_2, R_3\}$. Thus, P is used to obtain an optimal plan for the query although it was not optimal for the subquery. In order to take into account the violation of principle of optimality due to the effect of ordering on cost, the notion of *interesting orders* was proposed in [S*79]. An order is *interesting* if it is on grouping or join columns of the query since such orders may be useful in a future join or a group-by operation. The only interesting orders that are generated are those that are due to choice of a join method (e.g., sort-merge) or existing physical access paths. Thus, an optimizer generates only a small number of interesting orders. Cost comparison takes place among plans with the *same interesting orders*. Since a suboptimal plan with an interesting order may result in a future optimal plan, an *unique* optimal plan for *every* generated interesting order is retained. Also, if some plan with an interesting order is cheaper than the plan with no interesting orders, then the latter is discarded.

4.2 Execution Space of Extended Left Deep Trees

In order to incorporate the transformations involving group-by, the optimizer considers the space of extended left-deep trees as its execution space where group-by as well as join may occur as internal nodes. Unfortunately, the resulting execution space is considerably larger than the traditional execution space of left-deep trees with interesting orders, for the following two reasons.

First, In addition to ordering the joins, the optimizer must also decide the placement of group-by operators. As a consequence, two plans (extended left-deep trees) over the same set of relations may now differ not only in the interesting order, but also in the sequences of group-by operators that have been applied. For example, a plan for $\{$`Order`, `Product`, `Division`$\}$ is incomparable to a plan obtained for the join between `Division` and the relation obtained by application of a group-by following the join of `Order` and `Product`. Therefore, over the same set of relations and for the same interesting order, more than one optimal plans need to be stored. A better understanding of the space of alternative executions can be gained by considering the representation of plans. Unlike the traditional case, a query can no longer be viewed as a set of relations to be linearly ordered. However, simple coalescing transformations have the property that if more than one group-by operators are applied (with intervening join nodes) then the resulting relation is the same as obtained by applying simply the last group-by operator[2]. Therefore, for optimization over extended left-deep trees, a query (or a subquery) can be viewed as a list (R, S) where R and S are sets of relations such that the *last* group-by operator was applied after the join of all relations in R and subsequently, the resulting relation was joined with all relations in the set S. A special case is $(R, \{\})$ where the group-by is applied after joining all relations in R.

Next, the space of potential interesting orders is also much larger compared to the traditional case if group-by operators are implemented using *sorting* (see Section 2.2). Intuitively, it is because orders that reduce the cost include not only the orders useful for the remaining join predicates and grouping columns of the

[2] Although the resulting relations are the same, the cost of the two plans may be different.

query, but also grouping columns (in all possible major to minor ordering) of possible group-by operators that may follow the subplan. To understand why that is so, let G_1 and G_2 be two group-by operators, implemented using sorting, that are applied one after another with intervening join operators that preserve the order created by G_1. If the choice of major to minor ordering of grouping columns for sorting in G_1 is such that it ensures that the stream is also sorted on the grouping columns of G_2, then the cost of evaluating G_2 is simply that of coalescing and evaluating aggregates, and no sorting is needed to form groups. Thus, the space of all possible major to minor orderings of all possible group-by operators is potentially large. These two factors make the cost of any simple-minded enumeration of extended left-deep join trees prohibitive. We defer a complete analysis of the effect of the above two factors on the size of the execution space to the full paper [CS94].

4.3 Greedy Conservative Heuristic

The optimization technique that we propose is *greedy* in that it applies group-by if it yields a better plan locally. The technique is *conservative* from the point of view of the time it spends in optimizing a query.

The *greedy conservative* heuristic places a group-by preceding a table scan or a join *if and only if* it results in a cheaper plan for *that* scan or join. In other words, we modify Step 2 in *Enumerate* to construct optimal plans for the following three cases and the plan with the least cost among them (for the same interesting order) is chosen: (a) $(S_j, \{R_j\})$, i.e., an application of a group-by to relation S_j prior to the join with R_j. (b) $(\{R_j\}, S_j)$, i.e., an application of a group-by operator on R_j prior to the join with S_j (c) Step 2 of *Enumerate*, i.e., no application of group-by at all, but simply taking the join of R_j with S_j. (Incidentally, the optimizer has to check also whether plans (a) and (b) are semantically correct. For example, both (a) and (b) can not be simultaneously correct if only the simple coalescing transformation is used.) Observe that the above *greedy* strategy has a very significant payoff from the point of view of reducing optimization cost. By *locally* choosing between the use and nonuse of group-by, a query can be viewed as a *set* of relations for which there is a *unique* plan for every interesting order. This is no different from the traditional case!

To complete our description of the optimization algorithm, we also need to specify what major to minor orderings of grouping columns will be considered by the optimizer in alternatives (a) and (b) above if a sort-based implementation of group-by is used. In greedy conservative heuristic, we have chosen to generate a *single* major to minor ordering for the group-by operator. The ordering is determined by the method used to join R_j and S_j:

1. For sort-merge, choose a major to minor ordering that is same as that for the join node.
2. Otherwise, if the group-by is on the outer relation of join, then choose grouping columns of the query to be the major sort columns.

We consider (2) since many join methods (e.g., nested loop) preserve the ordering of the outer relation. If all subsequent joins preserve the order introduced by (2), then no sorting is needed in the remainder of query evaluation. If neither (1) nor (2) is true, then the choice of major to minor ordering may be determined by cost of sorting, or, in the absence of a detailed cost model, an arbitrary order is chosen. Our strategy for picking the major to minor ordering leads to very *conservative* increase in the number of interesting orders generated. Note that interesting orders in (1) are also generated by the traditional optimizer. Therefore, our optimizer may generate *at most one* additional interesting order only (if (2) is applicable). Thus, we have also avoided the problem of generating too many interesting orders.

Example 4.1: Consider the following SQL query:

```
Select Sum(R.a)
From R,S,T
Where R.b = S.b and S.c = T.c
Group By R.d
```

The simple coalescing transformation enables pushing group-by to the relation `R` prior to doing the scan with grouping columns {`R.b`, `R.d`}. While considering the join between `R` and `S`, the choice between evaluating or not evaluating an early group-by will be considered and the cheapest plan will be retained for joining with `T`. Let us now consider the plans that are generated by the greedy conservative heuristic while considering the join between `R` and `S` with an early group-by. Assume that nested loop and sort-merge are the only two methods. In that case, we will consider the major to minor ordering `(R.d, R.b)` for nested loop and `(R.b, R.d)` for sort-merge. These will be the candidate plans with early group-by. ▮

We have discussed how greedy conservative heuristic ensures that the search space is reasonable. In the following section, we will present experimental study that indicates that the increase in the optimization cost is indeed modest. For a detailed analysis, see [CS94]. We now discuss two other important aspects. First, we explain why for I/O-based cost models, the plan produced by greedy conservative is *never* more expensive than those obtained by using traditional algorithm. Moreover, experimental results indicate that the extent of improvement in the quality of plans is significant. Next, we show how we can integrate greedy heuristic with a System-R style optimizer.

Optimality: While the formal treatment of optimality results on greedy conservative heuristic appears in [CS94], we will sketch the intuitive proof why the plans obtained by greedy conservative are no worse than those obtained by the traditional optimization algorithm for cost models based on I/O cost. To be precise, we can show the above result for cost models that satisfy the following property:

- For the same interesting order, the cost of a group-by is a monotonic function of only the size of the relation to be grouped.

There are two observations that hold the key to optimality. First, by definition, the greedy conservative heuristic ensures that if we choose to do an early group-by, then the local cost of the application of group-by followed by the join (or scan) is no more than doing only a join. Next, application of a group-by never increases the size of the relation. Therefore, it follows from the assumption we have just made about the cost model that the greedy conservative heuristic never adds to the cost of the group-by at the final step (if any). Finally, note that the greedy conservative heuristic retains all the interesting orders generated by the traditional optimizer. Therefore, the plan chosen by the greedy conservative heuristic is never worse than the traditional plan.

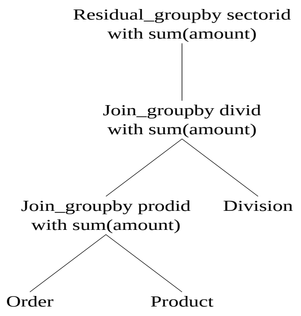

Figure 4: Implementation of Group-By

Ease of Integration: For incorporating the greedy conservative heuristic, we needed only modest extensions to our existing optimizer since the option of an early group-by can be considered in conjunction with choice of access methods and join algorithms. We view the group-by operator that precedes join as a new access path that needs to be considered. Thus, we model the join and the preceding group-by as a single operator `Join_groupby`. Such an operator will be considered during Step 2 of *Enumerate*. However, there may still need to be a `Residual_groupby` operator that succeeds all the join operators. Thus, the extended left-deep tree in Figure 2 (Plan 2) can be represented in terms of the above two operators as in Figure 4. The benefits of the above style of modeling is that the extensions are similar to that needed for adding alternative implementations for join and scan. Finally, in order to predict the costs of alternatives correctly, the implementation detects cases where a group-by is redundant (Section 3.4) and cases where a set of group-by nodes introduced by simple coalescing satisfies invariant grouping (Section 3.2).

4.4 Cost Models

Note that the transformations presented in this paper are not dependent on the specificity of the cost model. Also, the only extension to the traditional cost model that we need is in estimating the number of tuples of the relation after a group-by. For a group-by with a single grouping column, the number of groups is equal to number of distinct values in that column. However, for multi-column group-by, we need to develop an estimate for the number of groups. Since we wanted to be conservative in pushing an early group-by, we adopted a cost model that discourages early grouping by providing a *guaranteed overestimation* of the number of groups:

- It is assumed that the grouping columns are independent. The number of distinct values are estimated to be the minimum of the following two quantities: (a) Number of tuples in the relation (b) Product of the number of unique values in the grouping columns.

Any optimal plan obtained using this cost model can perform *no worse* than a traditional plan. Therefore, the choice of such a cost model is justified. However, by being less conservative, more realistic estimates of the number of groups are possible. An example of such a cost model is where we assume that the number of distinct values equals the maximum value of the cardinality among all the grouping columns. In other words, we assume that the columns have a "hierarchical" ordering.

5 Experimental Study

In the last section, we have shown that the *greedy conservative* heuristic never does worse than the traditional optimizer. In order to get a sense of how much better it does in terms of the *quality of plans* and what the corresponding *optimization overhead* is, we did an experimental study. We achieved statistically significant improvement in the quality of plans with a modest increase in the optimization cost. The experiments were conducted using our current implementation of the optimizer. The optimizer considers several join methods (nested-loops, merge-scan, and

	Comparison of Quality of Plans				Optimization Overhead	
No of joins	Av. Ratio (total)	Av. Ratio (only diff)	Max Ratio	% of diff. plans	Av. # of plans stored by traditional optimizer	Av. # of plans stored more than traditional optimizer
1	1.25	1.82	6.00	31%	3	0
2	1.31	1.67	5.06	46%	10	1
3	1.37	1.68	8.70	55%	29	2
4	1.48	1.76	11.40	64%	73	6
5	1.56	1.87	27.73	64%	156	17
6	1.69	2.03	19.87	67%	310	48

Figure 5: Study of Performance Metrics

simple and hybrid hash-join). A detailed description appears in [CS94].

5.1 Framework

The conventional and the modified optimization algorithm were executed on queries with group-by clause consisting of equality joins. The sizes of these queries, which were generated randomly, ranged from one to six joins. Among all attributes participating in the query, 10% of attributes were chosen randomly and made either group-by attributes or aggregation attributes. At least one attribute was assigned as the grouping and another as the aggregation column of the query. We borrowed the experimental framework from [IK90, INSS92, K91] and we review some of the important details of that framework here. A randomly generated relation catalog where relation cardinalities ranged from 1000 to 100000 tuples was used. The number of unique values in each column was between 10% to 100% of the cardinality of that relation. Each relation had four attributes and one attribute was randomly chosen to be the primary key for each relation. Either a relation was physically sorted, or there was a B^+-tree or a hashing primary index on the key attribute. For each nonkey attribute, there was a 50% probability of having a secondary index on that attribute. In our experiment, only the cost for number of I/O (page) accesses [CS94, IK90, K91] was accounted.

5.2 Performance Metrics

For each choice of the query size and the parameters of the experiment, we chose to report the following four quantities as indexes of comparison of the quality of plans produced (Figure 5).

The *average ratio* represents the average of the ratio of cost of the optimal plan produced by the traditional optimizer and the cost of the plan obtained by our extended optimization algorithm. Thus, a factor 1.37 indicates that on average, the plan produced by the traditional optimizer costs 37% more than the cost of the plan obtained from the modified optimizer.

The *% difference* tells us how often the plan produced by the modified optimizer was better than the plan produced by the traditional optimizer. Observe that our result of Section 4 assures us that the greedy conservative is *never worse*.

The *average ratio (diff)* metric is similar to average ratio, except that we compute the average only over data points where the traditional and the modified algorithms differ. Intuitively, this metric represents the average extent of benefit when the modified optimizer produced a different plan.

The *max ratio* parameter measures the largest difference between the optimal plan of the traditional optimizer and the modified algorithm. Thus, it is maximum value among the ratios of cost of optimal plans of the traditional to that of our modified algorithm.

The other factor that we studied is the relative increase in the *cost of optimization*. For this comparison, we have presented two parameters: average number of plans stored in the traditional algorithm, and average number of more plans that were stored by the modified algorithm.

5.3 Observations

We generated queries from one to six joins with our experimental set-up. For each kind, 1500 or more queries were generated. We ensured that the enough number of trials were carried out so that the *average ratio* parameter was estimated with 5% error with a confidence of 95%. The following observations were made:

(1) The average ratio and average ratio (diff) indicated that our plans were significantly better than the traditional algorithm. (2) The number of plans maintained by the modified algorithm was modest. (3) Finally, the results show that there is a significant benefit of incorporating the transformation even when we assumed the conservative cost model which discourages group-by.

6 Conclusion

In this paper, we have presented new transformations that make it possible to push group-by past join operations. The advantage of early group-by is possible reduction in the sizes of the relations. We have presented three transformations of increasing generality and have shown their soundness. We observed that the proposed transformations must be applied in a cost-based fashion and applications of such transformations may influence the join order. Therefore, we addressed the question of how the transformations can be incorporated in System-R style dynamic programming algorithm, which is used widely in commercial optimizers. We proposed a *greedy conservative* heuristic that constrains the search space while ensuring that the enumeration based on the heuristic never arrives at a plan that is worse than the one produced by the traditional optimizer. Experimental results indicate that the plan chosen by our modified algorithm outperforms plans produced by the traditional optimizer without significant optimization overhead. Our solution also provides a simple optimization technique to push down duplicate elimination for `Select Distinct` queries and to ensure that the plans obtained by pushing down duplicate elimination are no worse than the traditional plans.

We have provided techniques to process and optimize single block queries with aggregates and grouping that are practical, easy to adopt and yet efficient. For the in-house data warehouse applications, such techniques helped processing of queries significantly.

7 Acknowledgement

The optimization and the processing issues in data-warehouse application were brought to our attention by Mark Sturdevant of Hewlett-Packard Corporate Data Management group. The discussions with and suggestions from Umesh Dayal were very useful. Thanks are due to Waqar Hasan, Ravi Krishnamurthy and Marie-Anne Neimat for their help.

References

[CLR90] Cormen T., Leiserson C., Rivest R.L. *Introduction to Algorithms*, MIT Press, 1990.

[CS94] Chaudhuri S., Shim K. "The promise of Early Aggregation," HPL Technical Report, 1994.

[DD93] Date C. J., Darwen H. "A Guide to the SQL Standard: A User's Guide," Addison-Wesley, 1993.

[DGK82] Dayal U., Goodman N., Katz R. H., "An Extended Relational Algebra with Control over Duplicate Elimination," *Proc. of the First ACM Symposium on Principles of Database Systems*, pp. 117-123, 1982.

[D87] Dayal U. "Of Nests and Trees: A Unified Approach to Processing Queries that contain subqueries, aggregates and quantifiers," in *Proceedings of the 13th VLDB*, Aug 1987.

[GHK92] Ganguly S., Hasan W., Krishnamurthy R. "Query Optimization for Parallel Execution," in *Proceedings of the 1992 ACM-SIGMOD Conference on the Manage ment of Data.*

[G87] Ganski R. A., Wong H. "Optimization of Nested Queries Revisited," in *Proceedings of 1987 ACM-SIGMOD Conference on Management of Data*, San Francisco, May 1987.

[INSS92] Ioannidis Y., Ng R., Shim K., Sellis T., "Parametric Query Optimization," in *Proceedings of the 18th International VLDB Conference*, Vancouver, Canada, August 1992

[IK90] Ioannidis Y., Kang Y., "Randomized Algorithms for Optimizing Large Join Queries," in *Proceedings of the 1990 ACM-SIGMOD Conference on the Management of Data*, Atlantic City, NJ, May 1990.

[ISO92] ISO. *Database Language SQL ISO/IEC*, Document ISO/IEC 9075:1992. Also available as ANSI Document ANSI X3.135-1992.

[K91] Kang, Y., "Randomized Algorithms for Query Optimization", Ph.D. Thesis, University of Wisconsin, Madison, WI, April 1991.

[K82] Kim W. "On Optimizing an SQL-like Nested Query," in *ACM Transactions on Database Systems*, 7(3):443-469, Sep 1982.

[Kl82b] Klug A. "Access Paths in the ABE Statistical Query Facility," in *Proceedings of 1982 ACM-SIGMOD Conference on the Management of Data.*

[LCW93] Lu H., Chan C. C., Wei K. K. "A Survey of Usage of SQL," SIGMOD Record, Vol 22, No. 4, 1993.

[M92] Muralikrishna M. "Improved Unnesting Algorithms for Join Aggregate SQL Queries," in *Proceedings of the 18th VLDB*, 1992.

[PHH92] Pirahesh H., Hellerstein J. M., Hasan W. "Extensible/Rule Based Query Rewrite Optimization in Starburst," in *Proc. of the 1992 ACM SIGMOD Conference on Management of Data*, June 1992.

[S*79] Selinger P. G. et.al. "Access Path Selection in a Relational Database Management" in *Proc. of the ACM SIGMOD Conference on Management of Data*, June 79, pp.23-34.

[TM91] Tsang A., Olschanowsky M. "A Study of Database 2 Customer Queries," IBM Santa Teresa Laboratory, TR-03.413.

[YL93] Yan W. P., Larson P. A., "Performing Group-By before Join," International Conference on Data Engineering, Feb. 1993, Houston.